Dynamic Link Library Dependency Tools

An Application Guide

Table of Contents

Introduction

Porting applications requires identifying all of the Dynamically Linked Libraries (DLLs) that an application references. Applications often reference DLLs that are not directly provided by the application's Independent Software Vendor (ISV). In some cases, ISVs are not fully aware themselves of all the externally controlled DLLs that their own applications require.

This document briefly describes the history and motivation of implicit and explicit DLL linking and lists various tools available that can help ISVs identify DLLs that are directly or indirectly referenced by their applications.

Implicit and Explicit Linking using DLLs

Most applications use routines exported from DLLs by using implicit linking with a supplied import library or by explicitly linking/loading a DLL at run-time using OS support calls, such as LoadLibrary.

Typically, DLLs that are implicitly linked into a process must be present on the system library search path at the time the process is loaded by the OS. All implicitly loaded DLLs must be loaded and mapped into a process space before the process entry point is called. This loading/mapping typically results in longer startup times, and in increased system resource usage.

In contrast, DLLs that are explicitly linked are usually loaded just before they are needed. Seldom needed DLLs or DLLs that provide specific features are typically loaded and unloaded based on run-time program conditions. Explicit linking allows the application to control the use of resources and deferring DLL loading until absolutely necessary.

Microsoft Visual C++* 6.0 and greater introduce a feature by which DLLs can be explicitly identified to the linker as being delay-loaded. This feature allows for the use of a traditional import library while allowing for DLLs to be loaded in a delayed fashion akin to explicit linking. Delayed-load imports can be viewed with dumpbin or link utilities (see dumpbin under Static Analysis Utilities). There is a wealth of information available on load-time and run-time dynamic linking in the Microsoft Platform SDK and MSDN* Library as well as many other third-party sources.

Methods of Static and Dynamic DLL Dependency Collection

Now we move on to the business of capturing DLL dependency information.

Gathering implicitly linked DLL dependency information for a process is achieved by iterating through the import table to find the initial set of DLLs that are implicitly linked. From this list of initial DLLs, their respective import tables are examined for the next set of implicitly linked DLLs, and so on, until all import tables have been searched and exhausted. This method is called static analysis, as it does not require the process in question to be running in order to collect implicit DLL information.

In contrast to static analysis, gathering explicitly linked DLL data for a process is a little more complex. In order to gather explicitly loaded DLLs, the process to be examined needs to be actively running. In this model, as a process loads DLLs explicitly (via LoadLibrary calls) at run-time, tools can be notified of the DLL load events, and the data captured. Alas, dynamic analysis is not 100% accurate unless all available application code paths can be guaranteed to be exercised.

A combination of static and dynamic analysis is key to identifying the majority of the DLL dependencies in application code.

The following section contains a listing of static and dynamic analysis utilities that are available today.

Static Analysis Utilities

Dependency Walker

The Dependency Walker (depends) utility is provided with the Microsoft Visual C++ Developer Studio*. Dependency Walker lists any implicitly linked DLLs referenced by the application that it can locate, as well as subordinate implicitly linked DLLs (if available). Dependency Walker displays import and export function names (and Ordinals if available) for each DLL, timestamp, size, and base load address information. This utility has a graphical interface.

Dumpbin

The dumpbin utility is provided with the Microsoft Visual C++ Developer Studio. Dumpbin can be used to dump the import table of portable executable images (dumpbin /imports <imagename>). Dependency walker does this job much more elegantly (by applying recursion). Dumpbin is a shim layer to the Microsoft Linker, which performs most of the hard work (equivalent invocation is link /dump /imports). This utility is command-line driven.

Dynamic Analysis Utilities

NTHandleEx

The NTHandleEx utility can be obtained from www.sysinternals.com. NtHandleEx displays all processes running on a system, so your application must be running. After selecting the process of interest, NtHandleEx lists all DLLs loaded by that process. The utility provides the fully qualified path name of the DLL or DRV, the version number (if available), the base address of the DLL as mapped in the process, and the DLL size. NtHandleEx provides a snapshot of the DLLs mapped in the process at a point in time, which may be different from the full list of DLLs used throughout the process' lifetime. This utility has a graphical interface.

ListDLL

The ListDLL utility can be obtained from www.sysinternals.com. ListDLL provides the same information listed by NtHandleEx, using a command-line invocation. Information is displayed to the console window. Use the command-line option, –p process_name, to display DLLs currently mapped in a specific process. If you do not specify this option, ListDLL displays DLL information for all processes running on a system.

Pview

The Pview utility is provided with the Microsoft Visual C++ Developer Studio. Pview displays all currently running processes on a system. After selecting the process of interest, Pview lists a snapshot of all DLLs mapped into that process space. Pview does not provide the fully qualified path name of the DLL or DRV. This utility has a graphical interface.

Vadump

The Vadump utility is provided with the Platform SDK available in the Microsoft Developer's Network (MSDN). Vadump produces a dump of the address space of a single process, in a manner similar to Pview. The output of Vadump goes to stdout, rather than to a separate output window.

Microsoft Developer Studio (Visual C++ IDE Shell)

When a process is loaded under the Microsoft Developer Studio under debug control, the output window displays the names of implicitly and explicitly loaded DLLs as they are mapped into the process.

WinDbg

WinDbg is the application debugger provided with the Microsoft Platform SDK available in the Microsoft Developer's Network (MSDN). WinDbg captures the name of each DLL that is mapped into a process under debug.

Gflags

The Gflags utility is provided with the Microsoft Windows Resource Kit available in the Microsoft Developer's Network (MSDN). Gflags allows the user to turn on the display of loader snaps. When loader snaps are enabled, as the loader loads DLLs into a process, the loader component displays diagnostic information about the DLLs as they are searched, loaded, and their initialization points called.